
Scheme in K

CS 422 – Fall 2006

Revision *Revision* : 1.0

Assigned November 27, 2006

Due December 15, 2006

1 Change Log

1.0 Initial Release.

2 Overview

In this unit project, you will define the core semantics of the Scheme language in both K and Maude. As you will see, there is a fair amount of flexibility in this – in some cases, you will be able to provide semantics directly for a language construct, while in others it may be best to transform it first into another construct. A parser for Scheme will be provided, so you will not need to worry about parsing it yourself.

3 Required Language Features

We will not be implementing all of Scheme, but just a subset. You are expected to implement the following features of the language.

- **Atoms:** You must provide support for basic atoms in Scheme. The required atoms are:

- Integers
- Booleans
- Strings
- Identifiers

```
> 5
5
> #t
#t
> #f
#f
> "hello, world!"
"hello, world!"
> a
10
```

You are not required to support the character type or the full Scheme numerical hierarchy, which includes rationals and reals as well. The above assumes that `a` is an identifier bound to the value 10.

- **Quote:** You must provide support for quoted expressions. Quotes are represented as either a single quote or the word `quote`; quoting prevents evaluation.

```

> 'a
a
> (quote a)
a
> (quote 5)
5
> '5
5
> '#f
#f

```

You can provide semantics just for `quote`, as use of the single quote is just syntactic sugar. Note that quoting constants (such as 5 above) does nothing, but quoting identifiers and lists (see below) prevents evaluation.

- **Lists:** You must provide support for lists, proper and improper. A proper list is defined inductively: a list is proper if it is empty or if its tail is a proper list. You also need to provide support for the list operations `cons`, `car`, and `cdr`, which are used to form lists by adding a new element to the front, return the head of a list, and return the tail of a list, respectively. Lists are made up of a sequence of expressions, and are heterogeneous – unlike in ML, elements of a list can have different types.

```

> (quote (1 2 3 4 5))
(1 2 3 4 5)
> (car (quote (1 2 3 4 5)))
1
> (cdr (quote (1 2 3 4 5)))
(2 3 4 5)
> (cons 1 (quote (2 3 4 5)))
(1 2 3 4 5)
> (cons 1 2)
(1 . 2)
> '(1 2 "hello" 4)
(1 2 "hello" 4)

```

- **Lambda:** You must provide support for the `lambda` keyword, which is used to create functions. The form of a lambda is `(lambda (ids) exps)`, where `ids` is a list of identifiers and `exps` is a list of expressions. There are two other cases we will not tackle; in the first, only one identifier is provided, but this is a list of arguments, while in the second `ids` is an improper list with the last element interpreted as a list of the remaining arguments.

```

> (lambda (x) (+ x 3))
#<procedure>
> (lambda (x y) (+ x y))
#<procedure>

```

- **Function Application:** In Scheme, unquoted lists are function applications. The first expression should evaluate to a procedure/function, while the remaining expressions are the arguments to the function. The examples above using `car`, `cdr`, and `cons` are examples of built-in functions. Note that function application is strict – you must evaluate the function expression and all argument expressions first, before applying the function to the resulting values. Also note that lexical scoping is used, so you will need to think about using closures.

```

> ((lambda (x) (+ x 3)) 5)

```

```

8
> ((lambda (x y) (+ x y)) 5 3)
8

```

- **Top-level Definitions:** The `define` keyword is used to bind a name to a value, which can be a constant value or a function. This changes the top-level environment, meaning the name is visible anywhere it is not shadowed by a `let`-bound name (see below). Specifically, one wrinkle in the lexical scoping is that free names in a function may be defined at the top level after the function is defined but before it is used, and new definitions will replace the values of old definitions. The name of the definition is available in the body, so top-level definitions of functions can use the function name in the function body.

```

> (define x 3)
> (define f (lambda (y) (+ x y)))
> (f 4)
7
> (define x 4)
> (f 4)
8
> (define l '(1 2 3 4 5))
> (car l)
1
> (cdr l)
(2 3 4 5)
> (cons 10 l)
(10 1 2 3 4 5)

```

- **If:** `if` provides a conditional with an optional else branch. Note that, like in C, any non-false value is considered true.

```

> (if #t "true")
"true"
> (if #f "true")
> (if (+ 3 5) "true" "false")
"true"
> (define div (lambda (n m) (if (eq? m 0) "Division by 0!" (/ n m))))
> (div 10 2)
5
> (div 10 0)
"Division by 0!"

```

- **Cond:** `cond` provides a multi-branch conditional, with each branch consisting of a condition and the expressions to be evaluated when the condition is true. Conditions are checked starting with the first, with an optional else branch which is executed when no conditions match.

```

> (define check0 (lambda (n)
  (cond ((> n 0) "Greater")
        ((< n 0) "Less than")
        (else "Zero"))))
> (check0 5)
"Greater"
> (check0 -5)

```

```
"Less than"
> (check0 0)
"Zero"
```

- **Let:** The `let` keyword allows for one or more bindings which will be used in the body of the `let`. The form of the `let` is `(let ((x e) ...) exps)`, where each `x` is bound to the value that results from evaluating the expression `e`. The bound names (the `x`'s) can be used in the expressions `exps`, but not in the expressions `e`. Note that `let` is syntactic sugar for an equivalent `lambda` expression; you can provide semantics either directly or by desugaring.

```
> (let ((x 3)) (+ x 5))
8
> (let ((x 3) (y 5)) (+ x y))
8
> (let ((x (* 5 8)) (y (/ 10 2))) (+ x y))
45
```

- **Letrec:** `letrec` is similar to `let`, excepting that the identifiers are available in the expressions which are evaluated to give them initial values. Specifically, this allows functions to call themselves recursively, and also allows for mutual recursion.

```
> (letrec ((fact (lambda (n)
                  (if (eq? n 0) 1 (* n (fact (- n 1))))))
    (fact 5))
120
```

- **Begin:** The `begin` keyword allows expressions and/or declarations to be sequenced. The value of the last expression in the list is the value of the `begin` expression.

```
> (begin (+ 1 2) (+ 3 4) (+ 5 6))
11
```

- **Assignment:** The `set!` expression allows values of identifiers to be changed. This differs from a `let` or `letrec` in that a new binding is not introduced.

```
> (define x 10)
> x
10
> (set! x 11)
> x
11
```

- **Call/cc:** `call/cc`, or `call-with-current-continuation` (the first is shorthand for the second), allows the current execution context to be saved and later retrieved. `call/cc` expects a single argument, a function of one parameter. This parameter is the current context, which is also represented as a function of one parameter. When it is invoked, the current context is restored, with the value passed used as the value of the computation. If it is not invoked, the computation continues normally. For instance, here is an example where the continuation is not used:

```
> (call/cc (lambda(k) (* 5 4)))
20
```

Here is an example where it is used, with the result as the result of the computation:

```
> (call/cc (lambda(k) (* 5 (k 4))))  
4
```

Here, the value passed is used as the value of the entire `call/cc`:

```
> (+ 2 (call/cc (lambda(k) (* 5 (k 4)))))  
6
```

These examples, and others, are available in §3.3 of the Dybvig text.

- **Built-in Functions:** You should provide semantics for the following built-ins:

- +
- -
- *
- /
- <=
- <
- >=
- >
- equal?
- null?
- number?
- procedure?
- string?
- boolean?
- and
- or
- not
- car
- cdr
- cons

You can assume that any of the operations above that can take two or more arguments only take two. For instance, `+` can take a list of numbers to sum, but you will only be given at most two. Note that `+` works with just one number as well, so `(+ 3)` evaluates to 3.